TOPIC 1

ELEMENTS OF JAVA NIO

NIO

JAVA NIO

- Alternative I/O Library for Java
- Released with Java 2 Version 1.4
- Different Programming Model
- *New I/O*

HISTORY OF NIO

- Java I/O that is introduced in 1996 in the very first version of the JDK.
- Java NIO has been added to the JDK in 2002, Java 2 version 1.4
- Java 7, Java NIO2 was introduced, bringing more advancements to `Paths`, `Files`

CONCEPTS FOR JAVA NIO

- Channels
- Buffers
- Selectors
- Pipes
- Paths
- Files

CHANNELS

- A channel represents an open connection to an entity such as:
  - Hardware device
  - `File`
  - A network socket
  - Program component that is capable of performing one or more distinct I/O operations

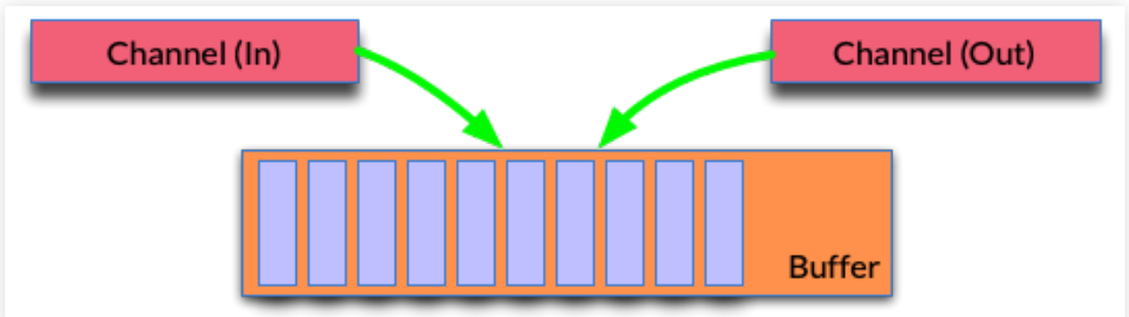
VARYING FLAVORS OF CHANNELS

- `FileChannel`
- `SocketChannel`
- `ServerSocketChannel`
- `DatagramChannel`
- `AsynchronousFileChannel`

BUFFERS

- Container of data for a specific data type
- Linear, finite sequence of primitives
- Storage off-heap
- Every primitive, other than `boolean` has a `Buffer`
  - `ByteBuffer`
  - `ShortBuffer`
  - `IntBuffer`
  - `LongBuffer`
  - `FloatBuffer`
  - `DoubleBuffer`
  - `CharBuffer`

RELATIONSHIP BETWEEN BUFFERS AND CHANNELS



INTERACTIONS WITH BUFFERS

1. Write data into the `Buffer`
2. Call `buffer.flip()`
3. Read data out of the Buffer
4. Call `buffer.clear()` or `buffer.compact()`

BUFFER INTERNALS

- Buffers keep track of how much you have written
- Buffers are off-heap and not subject to garbage collection
- Switch between writing and reading mode by performing a `flip()` of the buffer
- `flip()` places the pointer at the beginning of the `Buffer`
- `clear()` will clear the data
- `compact()` will clear the data you have read and place the unread data at the beginning.

BUFFER COMPONENTS

- **Capacity -** Number of elements it contains. Never Negative. Immutable
- **Limit -** Index of the first element that should **not** be read or written. A buffer's limit is never negative and is never greater than its capacity.
- **Position -** Index of the next element to be read or written. A buffer's position is never negative and is never greater than its limit.

ALLOCATING A BUFFER

Before starting to use a `Buffer` it must be allocated

```
ByteBuffer buf = ByteBuffer.allocate(1024);
```

WRITING TO A BUFFER

- Buffers can be written manually using `put`
- Can be read from a `Channel`

```
int bytesRead = inChannel.read(buf);
```

```
buf.put(127);
```

MORE ON `flip()`

- Changes from writing to reading mode
- Moves the `position` to `0` of the buffer
- Sets `limit` to where we finished writing
- `limit` is less than `capacity` which is the size of the buffer

`clear()`

- Clears the `Buffer`
- Prepares to write data into the `Buffer` again
- Sets `position` back to `0`
- Sets `limit` back to `capacity`
- Data that was not read is forgotten

`compact()`

- If there is still data in the `Buffer` that has not been read, you can call `buffer.compact()`
- Sets `position` to right after the last unread element
- Sets `limit` to capacity

`rewind()`

- `position` is set to `0`
- Any `mark` has been discarded
- Must set the `limit` accordingly, so as reads don’t "over-read" the `Buffer`

PIPES

- Java NIO Pipe
  - One-way data connection between two threads.
  - Pipe has a source channel and a sink channel.
  - You write data to the sink channel.
  - You read from the source channel.
  - Can be processed across `Thread` boundaries

CREATING A `Pipe`

```
Pipe pipe = Pipe.open();
```

WRITING TO A `Sink`

```
//initialize buffer, fill buffer, and flip to prepare

buf.flip();

while(buf.hasRemaining()) {
    sinkChannel.write(buf);
}
```

READING FROM A Source

```
Pipe.SourceChannel sourceChannel = pipe.source();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
```

DEMO: CHANNELS, BUFFERS, AND PIPES

In the *targeted_advanced_java/src/test/java* directory, navigate to the `com.evolutionnext.nio` package and open *ChannelTest.java*

`Files` AND `Paths`

- Part of the NIO update that makes it easier to perform I/O than in the old days with Standard I/O
- Added as part of Java 7

`Path`

- `Path` instance represents a path in the file system
  - Points to a file or directory
  - Can be absolute or relative
  - Obtaining the absolute path will have the full path

File

- Represents a file in the path
- Provides an API to manipulates files in the file system

`Paths` AND `Files` UTILITIES

- While `Path` and `File` represents the path and files respectively
- `Paths` and `Files` are objects with static methods to perform work

For example

```
Paths.get("/home/scott/myfile.txt");
Files.exists(path, new LinkOption[]{ LinkOption.NOFOLLOW_LINKS});
```

DEMO: `Paths` AND `Files`

In the *targeted_advanced_java/src/test/java* directory, navigate to the `com.evolutionnext.nio` package and open *ChannelTest.java*

SELECTORS

- With Selectors, we can use one thread with multiple channels
- Avoids context-switching between threads
- Register Multiple Channels with a `Selector` object
- I/O is performed on the channels
- Any channel we register with a selector must be a sub-class of `SelectableChannel`
- These are a special type of channels that can be put in non-blocking mode

CREATING THE `Selector`

```
Selector selector = Selector.open();
```

REGISTER ALL YOUR CHANNELS

- In order to use the channels, we register the channels with the selector
- It must be in *non-blocking-mode*
- `FileChannels` cannot be used since they do not support non-blocking-mode
- In the following, `SelectionKey.OP_READ` is an interest
- This returns `SelectionKey`

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

DIFFERENT EVENTS

There are four different even that we can subscribe to: * **Connect** – when a client attempts to connect to the server. Represented by `SelectionKey.OP_CONNECT` * **Accept** – when the server accepts a connection from a client. Represented by `SelectionKey.OP_ACCEPT` * **Read** – when the server is ready to read from the channel. Represented by `SelectionKey.OP_READ` * **Write** – when the server is ready to write to the channel. Represented by `SelectionKey.OP_WRITE`

INTEREST SET

```
int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept  = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE;
```

COMBINING INTERESTS WITH &

Determining what kind of events we want on what we want to watch

```
int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept  = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE
```

READY SET

- Defines the set of events that the channel is ready for
- Given the `SelectionKey` we can use that to determine if we are ready for a particular value
- We can conveniently use the following:

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWriteable();
```

OBTAINING THE CHANNEL

Given that we are responding to a behavior, we can object the `Channel`

```
Channel channel = key.channel();
```

We can also obtain the `Selector`

```
Selector selector = key.selector();
```

CHANNEL KEY SELECTION

- Programmatically we need to perform a continuous evaluation
- We must call `select()` in order to get what kind of call is being requested
    - This will block until a `Channel` is ready for use

```
int channels = selector.select();
```

ATTACHING OBJECTS

We can attach any `Object` to the `SelectionKey` to pass-along information * Custom ID * Extra Data

```
key.attach(Object);
Object object = key.attachment();
```

You can also attach when registering a channel

```
SelectionKey key = channel.register(
  selector, SelectionKey.OP_ACCEPT, object);
```

SELECTING BASED ON KEYS

Next we get an idea on what we need to process based on key

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

- Iterate over the `Set` and for each key
- Obtain the channel and perform any of the operations
- This must all be performed as a continuous loop
- You will need to implement your own close message, like poison pill to shut down the server

DEMO: CHANNELS, BUFFERS, AND PIPES

In the *targeted_advanced_java/src/main/java* directory, navigate to the `com.evolutionnext.nio` package and open *EchoServer.java* and *EchoClient.java*